How to Implement Context-Sensitive Help

BY RAY DUNCAN

As graphical user interfaces have become the standard, and on-line documentation has become a mandatory component of commercial software, developers have looked around for new ways to differentiate their products from those of their competitors. One of the more recent and clever innovations in the area of on-line help is *context sensitivity*. A software product with this feature responds to the user's request for assistance according to what the user is currently doing, instead of simply displaying the help file's main table of contents and expecting the user to navigate to the appropriate topic. For example, if the user presses the F1 key while the Save As dialog box is on the screen, the application will jump directly to a help screen that explains the constraints on filenames, how to change directories within the dialog, and so on.

When you first consider adding context help to your own program, you'll probably guess (as I did) that a first approach to solving the problem is simply keeping track of the mouse pointer and responding to WM_CHAR or WM_KEYxxx messages based on that position. However, a few moments of additional musing on the common Windows user interface widgets will quickly convince you that the problem must actually be more complicated. For example, the grunt work of displaying a menu pop-up on the screen, tracking the mouse within the pop-up, highlighting the menu item under the mouse pointer, and decoding the user's menu selection is all taken care of magically by USER.EXE; from the application's point of view, all it knows is that it gets a WM_INITMENU message before the menu is displayed and a WM_COMMAND message after the user has made a selection. Dialog handling, which allows for arbitrary combinations and groupings of controls, is even more magical.

As it turns out, there's more than one way to solve the context-sensitivity help problem. Windows is an exceedingly rich, complex environment, and the programmer who really wants to peek inside the environment's handling of menus and dialogs can find several ways to subvert these ordinarily atomic operations. The approved method, however, is for the application to register a Windows *hook*. A hook is nothing more than a callback procedure within the application that is entered by Windows when certain events occur; the application can filter the events by ignoring them, altering them, consuming them, or responding to them as it sees fit. Many classes of Windows applications, ranging from word processors to testing programs to debuggers, rely on the Windows support for hook procedures.

The multistep process of adding context-sensitive help to your Windows applications is described and illustrated with sample code.

Before we go any farther, I should mention that there are two parallel APIs in Windows for the installation, processing, and deinstallation of Windows hooks. The Microsoft manuals and SDK example programs all focus on the older API, consisting of SetWindowsHook(), UnhookWindowsHook(), and DefHookProc(). However, in this column, I'm only going to demonstrate the use of the hook API functions that first appeared in Windows 3.1—SetWindowsHookEx(), UnhookWindowsHookEx(), and CallNextHookEx(). The newer functions have a much cleaner design and have the additional advantage that they are symmetric with the hook functions in the WIN32 API.

CONTEXT-SENSITIVE HELP Adding any form of context-sensitive help to your application is a multistep process where all the elements must be coordinated and work smoothly together. In this column, I'm only going to show you how to implement an extremely simplistic form of context-sensitive help to a Windows program. If you've used elaborate on-line help systems such as the one in Microsoft Excel 4.0, you can imagine that the code in such a program devoted to context-sensitive help alone might well outweigh the totality of the applications written by us humbler developers! I'll leave such sophisticated implementations to your own creativity and experimentation.

Assume that we are starting with a working but vanilla Windows application that has a Help pop-up on its menu bar and an accelerator table that equates the F1 key to the Help Contents menu item. When the user picks the Help Contents menu item or presses the F1 key, the application launches WINHELP.EXE with the name of its help file to display the help table of contents. We want to modify the application so that if the user presses the F1 key while a pop-up menu is active, the help viewer displays the help topic that describes the application's menu selections rather than the usual table of contents. We can reach this goal by installing a windows hook function that sees every message generated by the

user's interaction with the application. The hook function will monitor for an F1 keystroke while a menu is active and will react to the keystroke by sending a special message to the application's main message handler. The main message handler will then activate the help viewer and steer it to the appropriate topic.

The first step in the implementation of this simple context-sensitive help is to prepare the help file. For those occasions when we want to display a specific topic rather than the table of contents, we must give the viewer a way to associate integer values passed by the application in a WinHelp() call with the context strings that are used internally to the help file as the targets of hyperlinks. This association is accomplished by adding entries to the [MAP] section of the help project (.HPJ) file that controls the compilation of the ultimate help (.HLP) file. For example, imagine that we are building the file EXELOOK.HLP that contains the online documentation for the program EXELOOK.EXE. To assign the integer identifier 1 to the EXELOOK.HLP topic labeled by the context string menu_index, we would edit the following lines into the EXELOOK.HPJ file:

```
[MAP]
menu_index 1
```

and then rebuild the EXELOOK.HLP file with the command: HC EXELOOK. The next steps would be to add code to the initialization sequence of the application to register a private window message for use by the hook function with RegisterWindowMessage(), then allocate a thunk address for the hook procedure with MakeProcInstance(), and install the hook function with SetWindowsHookEx(). One of the parameters for SetWindowsHookEx() is a code that determines what type of events will be passed to the hook procedure; in this case, we'd use the code WH_MSGFILTER, which means we want our hook procedure to be notified for every message that is related to our application but not for any other type of "hookable" event. SetWindowsHookEx() returns a *hook handle* that must be stored in a global variable for later use. Of course, code must also be added to the program's cleanup sequence to remove the hook procedure from the chain of all such callbacks by calling UnhookWindowsHookEx() with the hook handle, and (optionally) to release the thunk address with FreeProcInstance().

Now we must write the hook callback procedure itself, using the guidelines in the Windows API function reference for the particular type of event filtering that our procedure is going to perform. A hook procedure is always called back by Windows with three parameters: an integer referred to as nCode that classifies the event, a 16-bit value whose meaning depends on the event type, and a 32-bit value whose significance also depends on the event type. In the case of message

Source Code Skeleton for Context-Sensitive Help

1 of 2

```
// Skeleton code for context-sensitive Help
// in a Windows 3.1 application.
// Copyright (C) 1993 Ray Duncan
// PC Magazine * Ziff Davis Publishing

// string used to allocate private message number
// for use by application's windows hook callback
#define HOOKMSGSTRING "HelpHookMessage"

// context ID for context-sensitive help. must be
// synchronized with the [MAP] section of HPJ file.
#define MENU_INDEX_HELP   1

// far pointer to windows hook callback
WNDPROC lpfnWindowsHookProc;

// handle for frame window
HWND hwndFrame = NULL;

// program instance handle
HANDLE hInst;

// handle for Windows hook function
HHOOK hWindowsHook;

//
// Table of window messages supported by FrameWndProc()
// and the functions which correspond to each message.
// The first slot is filled in with a private message number
// by InitInstance().
//
struct decodeWord messages[] = {
    0, DoHelpContext,
    WM_CREATE, DoCreate,
    WM_INITMENU, DoInitMenu,
    WM_SIZE, DoSize,
    WM_COMMAND, DoCommand,
    WM_CLOSE, DoClose,
    WM_DESTROY, DoDestroy, } ;

    .
    .
    .

//
// WinMain() is the usual application entry point
//
INT APIENTRY WinMain(HANDLE hInstance, HANDLE hPrevInstance,
                     LPSTR lpszCmdLine, INT nCmdShow)
{
    // save instance handle
    hInst = hInstance;

    // If this is the first instance of the application,
    // register window classes, or exit
    if(!hPrevInstance)
    {
        if (!InitApp())
            return(FALSE);
    }

    // Create the frame and do other instance-specific
    // initialization, or exit
    if(!InitInstance(lpszCmdLine, nCmdShow))
        return(FALSE);

    while(GetMessage(&msg, NULL, 0, 0)) // while msg != WM_QUIT
    {
        TranslateMessage(&msg);         // translate virt keys
        DispatchMessage(&msg);          // dispatch message
    }

    // clean up all allocations
    TermInstance();
}

//
// InitInstance() -- Performs a per-instance initialization
// of application, creating the frame window, allocating
// private messages, etc.  Returns TRUE if initialization
// successful, FALSE otherwise.
//
BOOL InitInstance(LPSTR lpCmdLine, UINT nCmdShow)
{
    .
    .
    .

    // allocate private message number for use by
    // windows hook callback procedure
    messages[0].Code =
        RegisterWindowMessage((LPSTR) HOOKMSGSTRING);

    if(messages[0].Code == 0)
        return(FALSE);

    // allocate thunk for windows hook callback
    lpfnWindowsHookProc =
        MakeProcInstance((WNDPROC) WindowsHookProc, hInst);
```

Figure 1: **This code outlines the procedures for implementing context-sensitive help in your Windows applications.**

Source Code Skeleton for Context-Sensitive Help

2 of 2

```
    // bail out if thunk allocation failed
    if(lpfnWindowsHookProc == NULL)
        return(FALSE);

    // register windows hook callback and save handle
    hWindowsHook =
        SetWindowsHookEx(WH_MSGFILTER, lpfnWindowsHookProc,
                         hInst, GetCurrentTask());
    .
    .
    .
    return(TRUE);
}

//
// TermInstance() -- global cleanup for application instance.
//
BOOL TermInstance(VOID)
{
    .
    .
    .
    // delete our windows hook function from global chain
    UnhookWindowsHookEx(hWindowsHook);

    // release the thunk for our windows callback
    FreeProcInstance(lpfnWindowsHookProc);

    return(TRUE);
}

//
// FrameWndProc() receives all messages for the frame window.
// It looks up the new message in the table messages[] and
// dispatches the appropriate routine if a match is found,
// otherwise it calls DefWindowProc().
//
LONG FAR PASCAL FrameWndProc(HWND hWnd, UINT wMsg,
    UINT wParam, LONG lParam)
{
    INT i;

    for(i = 0; i < dim(messages); i++)
    {
        if(wMsg == messages[i].Code)
            return((*messages[i].Fxn)(hWnd, wMsg, wParam, lParam));
    }
    return(DefWindowProc(hWnd, wMsg, wParam, lParam));
}
    .
    .
    .
//
// DoHelpContext() -- this routine is called by
// FrameWndProc() for processing of the private message
// sent by the windows hook procedure. It
// invokes the help file viewer with a context ID
// in order to deliver context-specific help.
//
static LONG DoHelpContext(HWND hWnd, UINT wMsg,
    UINT wParam, LONG lParam)
{
    WinHelp(hWnd, szHelpFileName, HELP_CONTEXT, MENU_INDEX_HELP);
    return(FALSE);
}

//
// WindowsHookProc -- windows hook callback function.
// This function must be registered with SetWindowsHookEx()
// and exported in the application DEF file.
//
DWORD FAR PASCAL WindowsHookProc(INT nCode, UINT wParam, LPMSG lpMsg)
{
    // bail out if callback unknown type or if pointer
    // to message structure is not valid
    if((nCode < 0) || !lpMsg)
        return(CallNextHookEx(hWindowsHook, nCode, wParam, lpMsg));

    // check if this is F1 keypress during a menu popup
    if(nCode == MSGF_MENU)
        if((lpMsg->message==WM_KEYDOWN) &&
           (lpMsg->wParam == VK_F1))
        {
            // yes, post private message number to
            // our frame window's message handler
            PostMessage(hwndFrame, messages[0].Code,
                        nCode, lpMsg->hwnd);
        }

    // pass message on to the other windows hooks procs in chain
    return(CallNextHookEx(hWindowsHook, nCode, wParam, lpMsg));
}
```

events, nCode specifies whether a menu or dialog is active, the other 16-bit value is unused, and the 32-bit value is a far pointer to the msg[] structure containing (among other things) the familiar values hWnd, wMsg, wParam, and lParam. We design our hook procedure to test nCode for the value MSGF_MENU, then access the msg[] structure via the 32-bit pointer and check for the message type WM_KEYDOWN and the virtual key value for the F1 key.

If all these conditions are met, we call PostMessage() with the private message number that was previously obtained from RegisterWindowMessage(). (Calls to SendMessage() should be avoided in hook functions because they can result in deadlock conditions.) Finally, we pass the event to the next Windows hook procedure in the chain by calling CallNextHookEx().

We're almost done now. We only need to add some lines of code to the message handler for our application's main window that will detect the private message number posted by the callback procedure. To process this message, we call WinHelp() with the command HELP_CONTEXT, the pathname for the help file, and the identifier that was associated with the context string for the menu topic in the [MAP] section of the HPJ file. The effect of this call is to launch WINHELP.EXE, if it isn't already running, and the menu topic is displayed in the main viewer window. The skeleton for all this context-sensitive help code can be found in Figure 1, and the source code for a version of EXELOOK.EXE that supports context-sensitive help can be downloaded from PC MagNet.

The final step is to export the windows hook procedure by adding its name to the EXPORTS section of the application's module definition (.DEF) file, and then rebuild the application.

READER FEEDBACK Many of you have written about my columns on Windows NT and the Windows help system. I've selected a few letters to publish here.

I appreciate your mentioning simtel in your April 13, 1993, article in PC Magazine *(titled "The Hazards of Exploring Evolving Environments"), but many users are likely to find the TOPS-20 file system daunting. Similarly, garbo is not a U.S. site, and access to it can be finicky. You might want to direct readers to the mirror site at Washington University in St. Louis, wuarchive.wustl.edu. The wuarchive.wustl.edu site mirrors not only simtel but a huge number of other sites, offering a wealth of programs. The MS-DOS partition from simtel is echoed in /mirrors/msdos. There are also MS-DOS archives located at archive.msdos.umich.edu.*

You might also wish to mention to users who don't have direct Internet access that a mail-based FTP (file transfer protocol) server, BITFTP, exists at Princeton University and can be reached via e-mail at BITFTP@PUCC.EDU; sending the command HELP in the body of an e-mail

message will return instructions to you on how to use the BITFTP service.

David O'Donnell
via Internet

In your April 13, 1993, column, you give some erroneous advice.

First, the instructions you supplied to use anonymous FTP to retrieve .ZIP files from the simtel20 archive will result in the mangling of whatever file is being downloaded. The simtel20 is one of the few anonymous FTP archives where the magic word for setting the file type for transfer is not *"binary." On simtel20 the file type* must *be declared as "tenex." The reasons for this are moderately arcane but there is no way around it.*

Secondly, you point to garbo.uwasa.fi as a popular mirror of the simtel20 site. The garbo.uwasa.fi site does have many of the same files as the simtel20 archive, but it is not a mirror site; it does not have all the same files and it does not have the same directory structure. In addition, if the majority of the PC Magazine *readership is in the U.S., steering readers to an archive site in Finland could result in costly international transfers.*

Two sites that are mirrors of the simtel20 archive are oak.oakland.edu and wuarchive.wustl.edu, and both are in North America.

John Schmid
via Internet

WINDOWS HELP *I'm glad to see someone spreading the word that Windows contains an excellent built-in viewing tool for on-line Help.*

It's unfortunate that the SDK documents the common-enough but arcane RTF file format the way it does. Microsoft could have made things much easier by documenting the Windows help authoring system in the SDK the same way it documents the multimedia viewer for the MDK (Multimedia Development Kit). The multimedia viewer (that Microsoft uses, with some extensions, for such projects as CineMania) is, after all, just a superset of the WinHelp development environment. (I suspect that Microsoft is trying to keep this a secret, and that we will see multimedia viewer as the on-line help viewing engine in future versions of Windows.)

In case you didn't know, there's an update to HC31.EXE posted on Microsoft's BBS. It's named HCP.EXE, dated April 17, 1992, and uses XMS so you can compile larger help files without the risk of getting out-of-memory errors. [This program is also available in the Microsoft Library Forum on CompuServe.—R.D.]

I hope your articles promote better on-line help. But frankly, I don't think we're going to see significant improvement until the development community acknowledges that on-line help files are technical documents just like printed manuals and, as such, should be written by the technical documentation department, not programmers whose primary responsibility needs to be with the code.

Wayne Hausmann
via Internet

Thanks for your recent article in the May 11, 1993, issue of PC Magazine. *I built my first Windows help application using WordPerfect for Windows, Version 5.2. I encountered some differences between the two word processors when I tried to map your Microsoft Word examples into working text:*

- *WordPerfect for Windows handily exports and imports text in RTF format. This seems to indicate that only one copy of a help file needs to be maintained; that the RTF format can be edited directly, and the native WordPerfect format can be forgotten. However, I have experienced loss of some formatting codes when importing RTF documents into the editor. Therefore I will still abide by your suggestion of editing in one format and exporting to the other.*
- *The # symbol may not be used to number footnotes unless the Layout/Footnote/Options/Numbering Method is set to* character. *Then a string of valid characters must be specified in the Layout/Footnote/Options/Characters field. For instance, the string "#*$K+" would allow you to specify context strings (#) as well as build tags, titles, keywords, and browse sequence numbers.*
- *Use Tools/Comment/Create in WordPerfect for Windows to insert invisible text for hypertext links. Unfortunately, WordPerfect for Windows inserts leading and trailing carriage returns around the comment and highlights it with a shadow box; despite its ungainly appearance, this works. The strikeout and redline font styles are interpreted as visible text, and they are not suitable for hypertext links.*

James McNamara
via Internet

The help authoring support tools in the MSDN CD-ROM that you mention in your first help file article (April 27, 1993) are now available on the Internet. The help compiler for 3.0 and 3.1 along with WHPE (Windows Help Project Editor), the WinWord macros, and other goodies can be found on ftp.cica.indiana.edu in /pub/pc/win3/uploads/what.zip (a 1.5MB file). The Windows Help Authoring Guide is on the same machine: /pub/pc/win3/programr/whag.zip contains this guide in WinWord documents, and /pub/pc/win3/programr/hag.zip contains the guide in the Help format. Each file is approximately 500K.

Tim King
via Internet

Clearly, there's a dearth of information about WinHelp. Which brings me to a plug for a book I coauthored along with Dave Farkas and Joe Welinske: Developing Online Help for Windows. *It is published by Howard W. Sams Co.*

While the Windows Help Authoring Guide does contain a ton of valuable information, it's not very process-oriented. For example, Microsoft gives the macro syntax, but doesn't really explain where to use macros, or the intricacies of doing something like dynamically changing the button binding in a help system.

In our book, we included more how-to information, a large section on help design principles, complete SDK-like documentation, a look at development methodologies, and reviews of the automated help tools currently on the market. For more info, see the WinHelp Forum on CIS for a help file containing a full outline and chapter summaries.

Scott Boggan
via CompuServe

THE IN BOX Please send your comments, suggestions, or questions to me at any of the following e-mail addresses:
PC MagNet: 72241,52
MCI Mail: rduncan
Internet: duncan@cerf.net □